Object-oriented Programming Laws for Annotated Java Programs

Gabriel Falconieri Freitas
Universidade de Pernambuco (UPE), Brasil
grff@dsc.upe.br

Márcio Cornélio
Universidade de Pernambuco (UPE), Brasil
Universidade Federal Rural de Pernambuco (UFRPE), Brasil
marciocornelio@acm.org

Tiago Massoni Rohit Gheyi
Universidade Federal de Campina Grande (UFCG), Brasil
{massoni, rohit}@dsc.ufcg.edu.br

Object-oriented programming laws have been proposed in the context of languages that are not combined with a behavioral interface specification language (BISL). The strong dependence between source-code and interface specifications may cause a number of difficulties when transforming programs. In this paper we introduce a set of programming laws for object-oriented languages like Java combined with the Java Modeling Language (JML). The set of laws deals with object-oriented features taking into account their specifications. Some laws deal only with features of the specification language. These laws constitute a set of small transformations for the development of more elaborate ones like refactorings.

1 Introduction

Software changes constantly due to maintenance that leads to correction of fails or just to improve functionalities. However, some changes can take place to achieve quality factors like reuse and legibility. In these cases, changes should not alter the software behavior but only its internal structure. Improving the internal software structure is an activity known as refactoring [8]. To avoid errors due to modifications, every change has to be done following a discipline.

This discipline can be achieved by programming laws, as guidelines to informal programming practices, establishing a basis for formal and rigorous program development. They are largely known for imperative programming [12, 19]. Also, functional programming and logic programming have a set of laws described by Bird and de Moor [2] and Seres [20], respectively. Laws of object-oriented programming have also been addressed in [3, 7, 6].

Design by Contract (DbC) [18] is a development methodology that aims at the construction of reliable object-oriented systems. Its basic idea is that a contract is established among classes of a system. In this way, software developers should formally specify what is required and ensured by methods and types. The Java Modeling Language (JML) [16, 14] is a notation for formally specifying the behavior of Java classes and methods.

The set of programming laws for object-oriented programming we have nowadays is designed for program transformation with no relation to specifications languages designed for DbC. Changes in specification usually should discharge code updates, maintaining the conformance between code and specification. On the other hand, changes in program code may require changes in specifications as the behavior implemented by code may diverge from the meaning of the original specification. For instance, moving a redefined method to its superclass can be illegal if this transformation causes weakening of

I. Mackie and A. Martins Moreira (Eds.): Tenth International Workshop on Rule-Based Programming (RULE 2009)
EPTCS 21, 2010, pp. 65–76, doi:10.4204/EPTCS.21.6

```
1  public class PositiveIntegerData {
2    //@ private invariant value.intValue() > -1;
3    private Integer value;
4    public PositiveIntegerData() { value = new Integer(0); }
5
6    /*@ requires newValue != null && newValue.intValue() > -1;
7      @ ensures getValue().intValue() == newValue.intValue();  @*/
8    public void registerValue(Integer newValue) { /* ... */ }
9
10   //@ ensures \result != null;
11   public /*@ pure @*/ Integer getValue() { /* ... */  }
12
13   /*@ requires getValue() != null;
14     @ ensures !(\result).equals(""); @*/
15   public String format() { /* ... */ }
16 }
```

Figure 1: Class `PositiveIntegerData`

pre-conditions and strengthening of post-conditions. Transformation of object-oriented programs with formal contracts has already been addressed in a rather informal way [10].

A catalogue of laws (primitive transformations) to deal with Java programs annotated with JML has been proposed in [9], which specifies about 80 laws. Here we present one law that deals only with JML specifications and two JML-aware Java laws that deals with attributes and methods, respectively. A law that only deals with JML can impose conditions only on JML elements present in the program, whereas JML-aware Java laws involve both JML and Java elements for stating conditions. The three laws we present here are catalogued in [9].

In this paper, we define laws (Section 3) of object-oriented programming for Java that are aware of specifications written in JML, which we describe in Section 2. The laws we present here and other ones present in a more comprehensive catalogue [9] were applied to refactoring a JML-specified version of a core module of a Manufacturing Execution System (MES) [22]. In Section 4, we present proof of soundness regarding the JML parts of two laws. We present an example of program transformation by means of laws in Section 5. Final remarks appear in Section 6.

2 The Java Modeling Language

The Java Modeling Language (JML) is a behavioral interface specification language (BISL) [16, 14] tailored to Java [11]. Thus, JML serves to describe names and static information that appear in Java declarations and how they act, how they behave. JML specifications are written in the form of *special annotation* comments that are inserted directly in source code of programs. These comments must begin with an at-sign (**@**) and can be written in two ways: by using **//@ ...** or **/*@ ... @*/**.

In Figure 1, we present the class `PositiveIntegerData` that represents positive integers. We introduce an instance invariant (Line 2), which is a predicate that is true in all visible states of objects of a class [16]. The invariant in the example has private visibility and establishes that the attribute `value` must always be greater than -1.

JML uses the **requires** clause to specify the obligations of the caller of a method, what must be true

Law 1 ⟨*move invariant to superclass*⟩

class B **extends** A {
  //@ **invariant** ψ_1;
  ads cnts mds
}
class C **extends** B {
  //@ **invariant** ψ'_1 && ψ_2;
  cnts′ mds′
}
	$=_{cds,Main}$	**class** B **extends** A {
		  //@ **invariant** ψ_1 && ψ_2;
		  ads cnts mds
		}
		class C **extends** B {
		  //@ **invariant** ψ'_1;
		  cnts′ mds′
		}

where

$\psi_2 \equiv$ **this instanceof** C ==> $\psi_{\texttt{inv}}$

provided

(↔) **super** does not appear in ψ_2.

(→) ψ_2 does not contain occurrences of model fields declared in C, nor uncast occurrences of **this**.

□

to call a method. For instance, the precondition of the method `registerValue` requires the value of the Integer object to be registered to be greater than -1.

A postcondition specifies the implementor's obligation, what must be true at the end of a method, just before it returns to the caller. In JML, the **ensures** clause introduces a postcondition. For instance, the Line 7 introduces a normal postcondition that asserts that the final value of the Integer object we register is the same as the one the method receives as argument. The JML modifier **pure** (Line 11) indicates that the method doesn't have any side effects and hence can appear in specifications. In JML, the keyword **also** indicates that a method is extending the specification it inherits from its supertype.

3 Laws

Our laws extend object-oriented programming laws from other works [3, 6, 5, 7, 17]. The laws are written in an equational style. Each side of the equation corresponds to a template of a well-formed program. Programming laws, in which left-hand and right-hand sides are related by equality, are a concise presentation of a pair of laws. These laws precisely indicate the modifications that can be done to a program, stating their corresponding side-conditions. In fact, to apply a law, it is necessary to check (syntactic or semantic) side-conditions to ensure that the transformation is behavior-preserving and also maintains its well-formedness. We consider that we are dealing with only one package and working in a limited open system [7], in which classes of our system can depend on external libraries.

In Java and JML context, we need to guarantee that source-code continues meeting its specifications written in JML, taking into account the semantics of JML specifications along with the notion of specification inheritance [13]. Here we present a law for invariants written in JML.

A JML-annotated Java program has the format *cds Main*, where *cds* is the set of all classes of the program and *Main* corresponds to the unique class in the program which has a static main method. We use *cnts*, *ads* and *mds* inside a class to represents the class constructors, attributes and methods, respectively. We have to emphasize that they may contain the specifications of each constructor or

Law 2 ⟨*move reference type attribute to superclass*⟩

```
class B extends A {
  ads cnts mds
}
class C extends B {
  /*@ nullable @*/ T a;
  ads' cnts' mds'
}
```

$=_{cds,Main}$

```
class B extends A {
  /*@ nullable @*/ T a;
  ads cnts mds
}
class C extends B {
  ads' cnts' mds'
}
```

provided

JML:

(←) $D.a$, for any $D \leq B$ and $D \nleq C$ does not occur inside specifications of cds, $Main$, $cnts$, $cnts'$, mds nor mds'.

Java:

(↔) T is not a primitive type.

(→) (1) a is not declared in ads; (2) The attribute name a is not declared by the subclasses of B in cds.

(←) $D.a$, for any $D \leq B$ e $D \nleq C$ does not occur in cds, $Main$, $cnts$, $cnts'$, mds nor mds'.

□

method. It is not necessarily Java code only, we can also have the corresponding JML specifications.

In the laws, we use $cds_1 =_{cds,Main} cds_2$ to denote the equivalence of sets of class declarations cds_1 and cds_2, where cds is a context of class declarations for cds_1 and cds_2. We need to stress that this definition takes into account only sequential programs. We write '→' to indicate the condition that need to be satisfied to apply a law from left to right. Likewise, we use '←' to indicate what has to be satisfied when applying the law from right to left. We use '↔ to indicate conditions that must hold in both directions.

The first law we present (**Law 1**) allows us to move an invariant ψ_2 from a subclass C to its superclass B. The invariant we want to move only refers to instances of C as we require the invariant to be applicable only to instances of class C. To apply this law in any direction, we require that calls to **super** do not occur in ψ_2, since after law application (in both directions) these calls may refer different elements. To apply this law from left to right, model fields cannot appear in ψ_2 and occurrences of **this** must be cast otherwise the elements they refer may not be visible.

Concerning the soundness of this law, we take in account the inheritance of specifications in JML [13], in which inherited invariants are conjoined with locally added invariants. On the left-hand side, the invariant ψ_2, which is present in class C, is inherited by the subclasses of C and holds for all subclasses. On the right-hand side of the law, the invariant ψ_2 is inherited by all subclasses of B besides those that are not subclasses of C. For those classes that are subclasses of B, but not subclasses of C, the invariant holds because for objects of these classes the antecedent **instanceof** C fails and the whole implication is true, not changing the meaning of any original local invariant that inherits ψ_2.

By using **Law 2**, we can move an attribute to a superclass if it is not already declared in the superclass and if it does not cause name conflicts. The application of **Law 2**, from right to left, allows us to move an attribute downward. In this case, we prevent access to the attribute by the expression **this**, and we allow only accesses to a by C or subclasses of C, including accesses that appear in specifications.

In **Law 2**, we consider only attributes whose type is a reference type. There is another law for moving an attribute of primitive type. The reason for having two disctinct laws for dealing with attributes of primitive and reference types comes from the **nullable** keyword in **Law 2**. In JML, any declaration (except for local variables) whose type is a reference type is implicitly declared to be non-null, except when one adorns the declaration with a **nullable** annotation [16]. Thus, by default, JML always checks if a not nullable attribute is null in all visible states of the class that declares it. When we move an attribute to a superclass, this is not aware about the newly moved attribute and, therefore, this action can cause a undesirable behavior. In fact, if one instantiates the superclass, JML will raise an invariant exception reporting that the new attribute is null. To avoid this, we force attribute nullability to move it up. Then, if one wants to move a non-null a attribute, one needs to introduce **nullable** annotation before moving it. An attribute can become nullable applying a law named *make attribute nullable*, not presented here. Remember that in Java only reference types can be null.

Law 3 allows us to move a redefined method from a class to its superclass. The proviso concerning **super** is needed because its semantics may be affected when we move it from a subclass to a superclass, or vice-versa. We can only move the specification of a method if it does not refer to model fields of the class in which the method is originally declared. Furthermore, **this** expressions may occur in the target method specifications only if they are cast. In fact, as in the law the method has default visibility, only non-private elements can be referenced in its pre- and postconditions. This is similar to Java: the **this** expression may appear in *mbody'* if they have a cast and they mention only non-private attributes or methods of class C. The right-hand side of **Law 3** introduces type tests in each one of the specifications. In this way we assure that the original pre- and postconditions of the redefined method of C will only be applied to callers that are instances of C or instances of any of C's subclasses.

Law 3 ⟨*move redefined method to superclass: overriden method with non-default specification case*⟩

```
class B extends A {
   ads cnts mds
   //@ requires ψ1;
   //@ ensures ψ2;
   rt m(pds) { mbody }
}
class C extends B {
   ads' cnts' mds'
   //@ also
   //@ requires ψ'1;
   //@ ensures ψ'2;
   rt m(pds) { mbody' }
}
```

$=_{cds,Main}$

```
class B extends A {
   ads cnts mds
   //@ requires θ1 && ψ1;
   //@ ensures θ1 && ψ2;
   //@ also
   //@ requires θ2 && ψ'1;
   //@ ensures θ2 && ψ'2;
   //@ also
   //@ requires θ2 && ψ1;
   //@ ensures θ2 && ψ2;
   rt m(pds) {
      if (!(this instanceof C))
      { mbody } else { mbody' }
   }
}
class C extends B {
   ads' cnts' mds'
}
```

where

$\theta_1 \equiv\ !(\textbf{this instanceof } C)$ and $\theta_2 \equiv \textbf{ this instanceof } C$

provided

JML:

($\leftrightarrow$) **super** does not appear in ψ_1' nor in ψ_2'.

($\rightarrow$) Both ψ_1 and ψ_2 do not contain occurrences of model fields declared in C, nor uncast occurrences of **this**.

Java:

($\leftrightarrow$) (1) **super** and private attributes do not appear in $mbody'$; (2) **super**.m does not appear in mds'

($\rightarrow$) $mbody'$ does not contain uncast occurrences of **this** nor expressions of the form $((C)$**this**$).a$ and of the form $((C)$**this**$).m(e)$ for any attribute a nor method m, in ads' and mds', respectively, with private visibility.

($\leftarrow$) $m(pds)$ is not declared in mds'.

□

4 Soundness

The proofs we present here are only concerned with the JML parts of the laws. Proving the Java part is difficult due to aliasing, which can lead to representation exposure problems, for instance. In JML, specifications present in a class are inherited by its subclasses, provided they are not private. This leads us to two concepts: join of specifications and specification inheritance.

4.1 Join of specifications

In a program written in Java and annotated with JML, classes inherit not only attributes and methods from superclasses, they also inherit specifications of invariants, methods, history constraints, and initialisation predicates [13, 15]. Concerning methods, a method specification may consist of several specifications cases, which are introduced by the use of clauses such as **requires**, **assignable**, **ensures** [16]. Each specification case has a precondition (the default predicate is *true*) that states when the corresponding specification case applies to a call. The keyword **also** joins specifications cases. When a precondition of a specification case holds, the corresponding postcondition must hold also. defined earlier in this section. The definitions we present here are taken from [15]. The notation $T \triangleright (pre, post)$ is related to a specification case of an instance method that type checks when its receiver (**this**) has static type T. It also type checks in contexts where **this** has some subtype of T. In what follows, we introduce the definition of joint JML method specifications [15].

Definition 1 ***(Join of JML method specifications)*** *Let $T' \triangleright (pre', post')$ and $T \triangleright (pre, post)$ be specifications of an instance method m. Let U be a subtype of both T' and T. Then the join of $(pre', post')$ and $(pre, post)$ for U, written $(pre', post') \sqcup^U (pre, post)$, is the specification $U \triangleright (p, q)$ with precondition p:*

$$pre \parallel pre'$$

and postcondition q:

$$(\backslash\textbf{old}(pre') \texttt{==>} post')\ \&\&\ (\backslash\textbf{old}(pre) \texttt{==>} post)$$

□

In Definition 1, the precondition of joint method specifications is their disjunction. The postcondition of the join is a conjunction of implications (written ==> in JML's notation), stating that when a precondition holds (in the pre-state), the corresponding postcondition must hold.

$ext\_inv^{B}_{LHS}$
$= $ [by Definition 2]
$\bigwedge\{added\_inv^{U} \mid U \in supers(B_{LHS})\}$
$= $ [by set theory]
$\bigwedge\{added\_inv^{U} \mid U \in ((supers(B_{LHS}) \setminus supers(A)) \cup supers(A)\}$
$= $ [by definition of conjunction]
$(\bigwedge\{added\_inv^{U} \mid U \in ((supers(B_{LHS}) \setminus supers(A))\}) \wedge (\bigwedge\{added\_inv^{W} \mid W \in supers(A)\})$
$= $ [by definition of added invariant in B_{LHS}]
$\psi_1 \wedge (\bigwedge\{added\_inv^{W} \mid W \in supers(A)\})$
$= $ [by Propositional Logic]
$\psi_1 \wedge true \wedge (\bigwedge\{added\_inv^{W} \mid W \in supers(A)\})$
$= $ [by Propositional Logic]
$\psi_1 \wedge (false \Rightarrow \psi_{inv}) \wedge (\bigwedge\{added\_inv^{W} \mid W \in supers(A)\})$
$= $ [by type test for object of type B]
$\psi_1 \wedge (this\ instanceof\ C \Rightarrow \psi_{inv}) \wedge (\bigwedge\{added\_inv^{W} \mid W \in supers(A)\})$
$= $ [by definition of added invariant in B_{RHS}]
$(\bigwedge\{added\_inv^{U} \mid U \in ((supers(B_{RHS}) \setminus supers(A))\}) \wedge (\bigwedge\{added\_inv^{W} \mid W \in supers(A)\})$
$= $ [by definition of conjunction]
$\bigwedge\{added\_inv^{U} \mid U \in ((supers(B_{RHS}) \setminus supers(A)) \cup supers(A)\}$
$= $ [by set theory]
$\bigwedge\{added\_inv^{U} \mid U \in supers(B_{LHS})\}$
$= $ [by Definition 2]
$ext\_inv^{B}_{RHS}$

Figure 2: Proof of **Law 1 -** case of object of exact type B

4.2 Specification Inheritance

Subtypes in JML inherit specifications, besides attributes and methods. First, we introduce some notation for type specification. For a type T, the invariant predicate declared in the specification of T (without inheritance) is denoted by $added\_inv^{T}$. For a method m declared in a type T, the notation $added\_spec^{T}_{m} = (added\_pre^{T}_{m}, added\_post^{T}_{m})$ is the join of the specification cases in type T for m. If m is declared in T with no specification and is not overriding any method, then $added\_spec^{T}_{m} = (true, true)$, which is the default specification in JML. We use $supers(T)$ to denote the set of all supertypes of T (including T) and $methods(\mathcal{T})$ to denote the set of all instance method names declared in the specifications of the types in a set $\mathcal{T}$.

Definition 2 ***(Extended specification)*** *Suppose T has supertypes $supers(T)$, which includes T itself. Then the extended specification of T is a specification such that:*
methods: *for all methods $m \in methods(supers(T))$, the extended specification of m is the join of all added specifications for m in T and all its proper supertypes*

$$ext\_spec^{T}_{m} = \bigsqcup^{T}\{added\_spec^{U}_{m} \mid U \in supers(T)\}$$

invariant: *the extended invariant of T is the conjunction of all added invariants in T and its proper supertypes*

$$ext\_int^{T} = \bigwedge^{T}\{added\_inv^{U} \mid U \in supers(T)\}$$

□

The definitions we present here were introduced in [15] and are the ones we use in this paper.

$$
\begin{array}{l}
\quad ext\_spec_m^{B_{LHS}} \\
= \text{[by Definition 2]} \\
\quad \bigsqcup_{LHS}^{B} \{added\_spec_m^U \mid U \in supers(B)\} \\
= \text{[by set theory]} \\
\quad \bigsqcup_{LHS}^{B} \{added\_spec_m^U \mid U \in (supers(B) \setminus supers(A)) \cup supers(A)\} \\
= \text{[by definition of join with respect to } B\text{]} \\
\quad (\bigsqcup_{LHS}^{B} \{added\_spec_m^U \mid U \in (supers(B) \setminus supers(A))\}) \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= \text{[by definition of join of specification cases for } B_{LHS}\text{]} \\
\quad (\psi_1, \backslash old(\psi_1) \Rightarrow \psi_2) \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= \text{[by Propositional Logic]} \\
\quad ((\psi_1 \wedge true), \backslash old(\psi_1 \wedge true) \Rightarrow \psi_2) \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= \text{[by type test for object of type } B\text{]} \\
\quad ((\psi_1 \wedge \neg(this\ instanceof\ C)), \backslash old(\psi_1 \wedge \neg(this\ instanceof\ C)) \Rightarrow \psi_2) \\
\quad \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= \text{[by Propositional Logic]} \\
\quad ((\psi_1 \wedge \neg(this\ instanceof\ C)) \vee false \vee false, \backslash old(\psi_1 \wedge \neg(this\ instanceof\ C)) \Rightarrow \psi_2) \wedge true \wedge true) \\
\quad \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= \text{[by type test for object of type } B \text{ and Propositional Logic]} \\
\quad ((\psi_1 \wedge \neg(this\ instanceof\ C)) \vee ((this\ instanceof\ C) \wedge \psi_1) \vee ((this\ instanceof\ C) \wedge \psi_1'), \\
\quad\quad (\backslash old(\psi_1 \wedge \neg(this\ instanceof\ C)) \Rightarrow \psi_2) \wedge (\backslash old((this\ instanceof\ C) \wedge \psi_1) \Rightarrow \psi_2)) \\
\quad\quad \wedge (\backslash old((this\ instanceof\ C) \wedge \psi_1') \Rightarrow \psi_2')) \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= \text{[by definition of join of specification cases for } B_{RHS}\text{]} \\
\quad (\bigsqcup_{RHS}^{B} \{added\_spec_m^U \mid U \in (supers(B) \setminus supers(A))\}) \sqcup^B (\bigsqcup^A \{added\_spec_m^W \mid W \in supers(A)\}) \\
= \text{[by definition of join with respect to } B\text{]} \\
\quad \bigsqcup_{RHS}^{B} \{added\_spec_m^U \mid U \in (supers(B) \setminus supers(A)) \cup supers(A)\} \\
= \text{[by set theory]} \\
\quad \bigsqcup_{RHS}^{B} \{added\_spec_m^U \mid U \in supers(B)\} \\
= \text{[by Definition 2]} \\
\quad ext\_spec_m^{B_{RHS}}
\end{array}
$$

Figure 3: Proof of **Law 3 -** case of object of exact type B

4.3 Proofs

Here we present proofs for **Laws 1** and **3**. Both proofs involve dealing with cases associated to the types of objects related to the classes that are emphasized in the laws. We present the proof for just one case of these laws. The conditions of the laws guarantee that both programs that appear in the laws are well-typed. Concerning **Law 2**, it is a law for attributes in which specification inheritance is not taken into consideration.

In Figure 2, we present the proof for the case of **Law 1** in which the we consider an object of exact type B. Notice that in **Law 1**, on the left-hand side, an object of exact type B has to establish the (added) invariant ψ_1. The added invariant is given by $\psi_1 \wedge (this\ instanceof\ C \Rightarrow \psi_{inv})$, on the right-hand side. For an object of type B, the type test is false and the whole implication results true. The whole effect is the same of the invariant of class B on the left-hand side.

The proof for the case of **Law 3** in which we consider an object of exact type B is presented in Figure 3. On the left-hand side of this law, the specification case for method m in class B has precondition ψ_1

```
1  public class EvenIntegerData {
2    //@ private invariant value.intValue() % 2 == 0;
3    //@ private invariant value.intValue() > −1;
4    private Integer value;
5    public EvenIntegerData() { value = new Integer(0); }
6    /*@ requires newValue != null;
7      @ requires newValue.intValue() % 2 == 0 && newValue.intValue() > −1;
8      @ ensures getValue().intValue() == newValue.intValue(); @*/
9    public void registerValue(Integer newValue) { /* ... */ }
10   //@ ensures \result != null;
11   public /*@ pure @*/ Integer getValue() { /* ... */ }
12   /*@ requires getValue() != null;
13     @ ensures !(\result).equals(""); @*/
14   public String format() { /* ... */ }
15 }
```

Figure 4: Class `EvenIntegerData`

and postcondition ψ_2. On the right-hand side, we enrich this specification case with type tests involving the class name C, but with no impacts for objects with distinct types from C. The other specification case for method m on the right-hand side also involves a type test, having no effects for classes other than class C.

5 Application

In this section, we present an example composed of excerpts of Java classes annotated with JML, as it is refactored by means of successive application of programming laws. Classes `PositiveIntegerData` (Figure 1) and `EvenIntegerData` (Figure 4) represent positive and even integers, respectively. These classes are part of a software that stores and manipulates instances of positive and even integers.

The class `EvenIntegerData` (Figure 4) can only hold even positive integers because of the invariant written in Line 2. And, to reinforce the invariant, pre-conditions of method `registerValue` guarantee that only even and positive values are allowed. These classes share methods that have the same functionality. Moreover, both have an attribute called `value` to save the integer value of the respective data. To improve the design of this program (e.g. by reducing the amount of duplicated code) and to accept new data types (e.g. odd integers), we need to change the program in a disciplined way. In what follows, we present a guideline that leads to the same structure as obtained by applying the refactoring Extract Superclass [8]. We do not present all derivation steps, we omit most of them to save space, but each step is accomplished by the application of a law. A detailed derivation, with all the steps and their corresponding laws, can be found elsewhere [9].

Our starting point is composed by the classes presented in Figures 1 and 4. We first introduce a new class (`IntegerData`) to be the superclass of the existing classes, by applying Law ⟨*class elimination*⟩, from right to left. Then, we change the superclass of classes `PositiveIntegerData` and `EvenIntegerData` to be `IntegerData`, by applying Law ⟨*change superclass: from Object to another class*⟩, from left to right. We prepare classes `PositiveIntegerData` and `EvenIntegerData`, for moving attribute, invariant, and methods. First, we move the common attribute `value` to the superclass. This is accomplished by the application of a sequence of laws, beginning with Law ⟨*change attribute visibility:*

```
1 public class IntegerData {
2   //@ protected invariant value.intValue() > -1;
3   protected Integer value;
4
5   // @ ensures \result != null; @*/
6   public /*@ pure @*/ Integer getValue() { /* ... */ }
7 }
```

Figure 5: Class `IntegerData`

from private to public⟩, which changes the visibility of the attribute `value` to public, then we change its visibility to default by the application of another law. In the sequence, we apply **Law 2** to move the attribute `value` from class `PositiveIntegerData` to class `IntegerData`. This is followed by the application of other laws to eliminate the attribute `value` from class `EvenIntegerData`. Then, we move common methods to the class `IntegerData`. First, we apply Law⟨*move original method to superclass*⟩ to move methods from class `PositiveIntegerData` to `IntegerData`; then we apply **Law 3** to move redefined methods from `EvenIntegerData` to `IntegerData`. Another law allows us to simplify conditional commands. After moving the attribute `value` and methods to the superclass `IntegerData`, we change the invariant of classes `PositiveIntegerData` and `EvenIntegerData` to be in the format required by **Law 1**. We apply **Law 1** twice, then we simplify the invariant in class `IntegerData` and change its visibility.

In Figure 5, we present class an excerpt of class `IntegerData` after the application of the programming laws that lead to the refactoring *Extract Superclass* [8]. The final version of the `PositiveIntegerData` and `EvenIntegerData` has no `getValue` method and does not have the common invariant (see Figure 5, Line 2) because, now, they belong to `IntegerData`.

In [9], we applied the laws proposed here along with others to refactor a core module of a Manufacturing Execution System (MES) [22] software, which formalizes methods and procedures of production in an integrated system and presents data in more useful and systematic way. To control and manipulate data dynamically and in a highly configurable way, the MES software is built on top of a Meta Data API. We have refactored a JML-specified version of the Meta Data API [9] by applying primitive transformations expressed by means of our laws. We applied our laws to refactor code and to accommodate new features. For instance, we eliminate duplicate code by extracting a superclass that abstracts the behavior of other classes present in the system. This is described by the *Extract Superclass* refactoring [8]. Other refactorings presented by Fowler [8] (for instance, *Replace Conditional with Polymorphism* and *Pull Up Method*) were also applied to the Meta Data API by means of the proposed laws.

6 Conclusion

Object-oriented programming laws were proposed by Borba *et al.* [3] for an object-oriented language [4]. They proposed laws for classes and commands; they also define a normal form for object-oriented programs written in their language along with a reduction strategy. They demonstrate that the set of laws is complete with respect to this normal form. Cornélio [6, 5] proves the laws with respect to a copy semantics [4] and formally justifies, by using programming laws and data refinement, refactoring practices documented by Fowler [8]. Silva, Sampaio, and Liu consider object-oriented programming laws in a language with a reference semantics [21], applying such laws to code refactoring. Duarte [7] adapts the programming laws initially proposed in [3, 5] for the Java programming and proposes other laws for

language features that are not present in the language used in [3, 5].

In this paper, we proposed laws for object-oriented programming in the presence of a behavioral interface specification language. In the laws that deal with source-doce, we treat the transformation considering the restrictions imposed by the specifications. These laws are based on programming laws from previous work [3, 5, 7] that does not consider specifications. We have considered laws that address only a subset of the JML's Level 0 constructs [16], specially for lightweight specifications. Nevertheless, our preliminary focus is to cover most of the JML constructs that form the core notation used in the design by contract methodology. We have also applied our set of laws for reducing a JML-specified Java program to a normal form [9] to address the relative completeness of the set of laws proposed.

With respect to the order of application of programming laws in program transformation process, distinct orders may lead to different results. The conditions for application of a programming law usually requires the application of other programming laws, defining that their applications are not commutative. To obtain the desired target program, a proper law application sequence must be established. Commonly used sequences of applications of laws can be registered as single transformation rules.

Concerning the refactoring process, we can view a refactoring as a target transformation that can be reached by the application of primitive transformations expressed by means of programming laws. The process of application of programming laws terminates when the desired structure is reached. Cornélio [6] presents refactorings as rules constituted by a pair of programs, similar to a law, but the transformation described is more complex than the one of a law. The program on the left-hand side presents the class or classes before rule application; the right-hand side presents the classes after rule application. Refactoring rules capture complex transformations. Here we have not presented refactoring rules as in [6], but the application of the programming laws we presented here and in [9] lead us to a result similar to that presented in [6], that is, a refactoring is derived by the application of programming laws. Although at this moment our work does not provide a way to transform programs mechanically, it offers a more reliable and extensible alternative to address behavior-preserving transformations like refactorings. Moreover, since some conditions present in the laws are related to logic proofs, it is necessary to use a theorem prover as an auxiliary tool.

Differently from laws that deal only with constructs of an object-oriented programming language, the presence of a behavioral interface specification language requires that we be aware of issues related, for instance, to the visibility of specification and code constructs, invariant preservation when introducing calls to super and changing a parameter type to a supertype requires introducing casts in occurrences of the parameter in specifications.

As future work, we intend to describe laws to support other JML clauses like **initially**, **constraint**, **represents**, and model fields. Also, we intend to work on proofs for the Java parts of the laws based on a reference semantics [1].

Acknowledgements

We are partially supported by the Brazilian Research Agency, CNPq, grant 484813/2007-2, and by the National Institute of Science and Technology for Software Engineering (INES[1]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08.

[1] http://www.ines.org.br

References

[1] A. Banerjee & D. A. Naumann (2005): *Ownership confinement ensures representation independence for object-oriented programs*. *Journal of the ACM* 52(6), pp. 894—960.

[2] R. Bird & O. de Moor (1997): *Algebra of Programming*. Prentice Hall.

[3] P. Borba et al. (2004): *Algebraic reasoning for object-oriented programming*. *Science of Computer Programming* 52, pp. 53–100.

[4] A. L. C. Cavalcanti & D. A. Naumann (2000): *A Weakest Precondition Semantics for Refinement of Object-oriented Programs*. *IEEE Transactions on Software Engineering* 26(8), pp. 713–728.

[5] M. Cornélio (2004): *Refactoring as Formal Refinements*. Ph.D. thesis, Universidade Federal de Pernambuco.

[6] M. Cornélio et al. (2009): *Sound refactorings*. *Science of Computer Programming* Available at `http://dx.doi.org/10.1016/j.scico.2009.10.001`.

[7] R. Duarte (2008): *Parallelizing Java Programs Using Transformation Laws*. Master's thesis, Universidade Federal de Pernambuco (UFPE).

[8] M. Fowler (1999): *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] G. R. F. Freitas (2009): *Refactoring Annotated Java Programs: A Rule-Based Approach*. Master's thesis, Departamento de Sistemas e Computação, Universidade de Pernambuco. Available online.

[10] M. Goldstein, Y. A. Feldman & S. Tyszberowicz (2006): *Refactoring with Contracts*. In: *AGILE '06: Proceedings of the conference on AGILE 2006*. IEEE Computer Society, Washington, DC, USA, pp. 53–64.

[11] J. Gosling et al. (2005): *Java Language Specification*. Addison-Wesley Professional, 3rd edition edition.

[12] C. A. R. Hoare et al. (1987): *Laws of programming*. *Commun. ACM* 30(8), pp. 672–686.

[13] G. T. Leavens (2006): *JML's Rich, Inherited Specifications for Behavioral Subtypes*. In: Z. Liu & J. He, editors: *ICFEM*, *Lecture Notes in Computer Science* 4260. Springer, pp. 2–34.

[14] G. T. Leavens & Y. Cheon (2005): *Design by Contract with JML*. Available online.

[15] G. T. Leavens & D. A. Naumann (2006): *Behavioral Subtyping, Specification Inheritance, and Modular Reasoning*. Technical Report 06-20b, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.

[16] G. T. Leavens et al. (2008): *JML Reference Manual*. Available online.

[17] T. Massoni, R. Gheyi & P. Borba (2008): *Formal Model-Driven Program Refactoring*. In: J. L. Fiadeiro & P. Inverardi, editors: *FASE*, *Lecture Notes in Computer Science* 4961. Springer, pp. 362–376. Available at `http://dx.doi.org/10.1007/978-3-540-78743-3_27`.

[18] B. Meyer (1992): *Applying design by contract*. *IEEE Computer* 25, pp. 40–51.

[19] C. C. Morgan (1994): *Programming from Specifications*. Prentice Hall, second edition.

[20] S. Seres (2001): *The Algebra of Logic Programming*. Ph.D. thesis, Oxfor University Computing Laboratory.

[21] L. Silva, A. Sampaio & Z. Liu (2008): *Laws of Object-Orientation with Reference Semantics*. In: *SEFM '08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society, Washington, DC, USA, pp. 217–226.

[22] R. R. Zagidullin & E. B. Frolov (2008): *Control of manufacturing production by means of MES systems*. *Russian Engineering Research* 28(2), pp. 166–168.